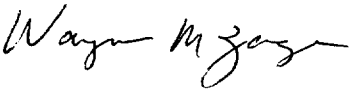
Algorithms Developed by Applying Design Metrics to Time-Slices during the Software Development Life Cycle

An Honors Thesis (HONRS 499)

by

Cathy Jeffers

Dr. Wayne Zage



Ball State University

Muncie, Indiana

April 1993

Graduation Date: May 1993

ACKNOWLEDGEMENTS

I would like to thank the members of the Design Metrics Research Team at Ball State University for all their help in the data collection. I would also like to thank Dolores and Wayne Zage, principal investigators on the Design Metrics Research Team for their assistance and advice. Special thanks goes to graduate student Raju Maharjan for all his hard work on the figures and tables he developed for this paper.

Table of Contents

ABSTRACT

The costs of software development are increasing, as is the use of software systems. Effective software engineering techniques are desired and needed to increase the reliability of software systems, to increase the productivity of the development team, and to reduce the costs of software development. To aid in meeting these needs, the Design Metrics Research Team at Ball State University is analyzing university and industry software to be able to make predictions of error-prone modules. The research team has developed and tested their design metrics and found them to be successful. This paper includes the study of one graduate university project and the conclusions drawn based on the accuracy of the design metrics applied to this project. Many algorithms were developed and tested, resulting in the creation of one algorithm that is a combination of the algorithms with the best prediction results. This new algorithm will be tested and evaluated in the study of the next project.

INTRODUCTION

Computer systems are being incorporated into all aspects of life and business. This means that personal, corporate, national and international economies are becoming increasingly dependent on computers and their software systems. Software costs are the major component of computer system costs, thus putting pressure on effective software engineering.

Developing and implementing software systems is a costly and time consuming endeavor. Lack of accurate estimation techniques can cause these systems to frequently be late and over budget. To counteract these negative characteristics, collecting metrics has aided in providing better software and better estimations of expenditures of time and money. A metric is a count on or a categorization of some aspect of a piece of software. The most common metrics used in software development are calculated from the source code, which is created during one of the latter stage of the development process. In these final stages, however, it is often too late to easily make changes. The earlier the problems can be detected, the easier they can be solved. "Defects introduced in the design phase lurk in the code, ready to bite you at the end, when they're most expensive to fix" (Keuffel 1990).

To learn how to identify problems early in the software development life cycle, the Design Metrics Research Team at Ball State University, Muncie, Indiana is collecting metrics at different time slices throughout the life cycle. It is important to see the changes made from one phase to another. Of all the life cycle phases, the design phase is of special interest. Here in this early phase, changes can be made at lower costs to the software engineer.

In the computer science curriculum at Ball State, there are three software engineering courses, one at the graduate level, and two at the undergraduate level. In these courses, students, documenting every phase of the software life cycle, develop software systems. The Design Metrics Research Team uses these projects for evaluation and study.

In the spring semester of 1992, thirty-six students registered for the graduate level software engineering course, CS 680. These students, forming twelve teams, were given the task of creating a consumer financial software system. The project, implemented in the language of each group's choice, was to be completed by the end of the semester. The requirements were the following:

REQUIREMENT SHEET

CS 680 Project--Spring Semester, 1992

You are to design, code, implement and document a Consumer Financial Software System (CFSS). This system is to be used by consumers to keep track of their finances and plan their financial future. The system will be menu driven and have the following options:

1. check register control

This component of the system should allow the user to

(a) deposit money into checking
(b) transfer money from savings to checking
(c) automatically add or deduct an amount on a certain day of the month
(d) categorize check expenditures into the following categories:

- household (rent or mortgage, utilities, etc)
- food
- business
- transportation
- medical/insurance
- charities
- entertainment
- other

(e) record check number, payee information and date
(f) mark checks as cleared or uncleared, and list outstanding checks
(g) provide yearly totals of deposits, withdrawals listed above
(g) obtain the balance available in checking

2. savings information

Allow the user to record

(a) certificate of deposit information, including term, interest rate, place of deposit, maturity date, and interest expected
(b) retirement account information, including term, pensions, social security, IRAs and SRAs
(c) day-in, day-out account information. In this user should be able to

- deposit money into, and withdraw money from savings
- transfer money from savings to checking
- obtain a yearly summary of deposits and withdrawals made
- obtain the balance available in savings

3. obligations

This part of the system should allow the user to list standing obligations, including mortgage or rent payments, car payments, credit card payments etc. For each of these, list the monthly payment, interest rate, and payoff date. Also give the total of the monthly obligations.

4. financial planners

(a) savings planner

This option allows the consumer to review various savings scenarios. He or she should be able to save a set amount of money each week, month or year,for a fixed length of time, at a fixed length of time, at fixed interest rate, and see that amount that would result in the account. Allow the user to try as many different scenarios as he or she wishes before exiting this option.

(b) borrowing planner

This option allows the consumer to review various borrowing scenarios.He or she should be able to determine the monthly payment when borrowing a set amount of money for a fixed length of time, at a fixed interest rate.Also allow the user to see the loan balance and the interest paid at yearly intervals. Again, the user should be able to try as many different plans as he or she wishes before exiting this option.

5. net worth calculator

Enter values from the CFSS (as much as possible) in the following categories,and then allow the consumer to supplement those values to help determine net worth:

- savings balances
- checking account balance
- stocks and bonds
- value of car
- value of home furnishings
- value of retirement accounts
- other assets

- amount owed on a house
- amount owed on a car
- amount owed on credit cards
- other liabilities

As many values as possible, such as the savings and checking amounts,must be entered automatically from the system.

6. personal budget planner

Using this option, the consumer can map out a monthly budget. Include at least the following categories, but feel free to add other categories to customize the planner to fit your own lifestyle

- housing
- Utilities
- Food
- Transportation
- Entertainment
- Medical
- Insurance
- Clothing
- Miscellaneous

7. credit card manager

The user can select this option to generate a list of credit card purchases that is categorized by the credit cards used.

8. financial statistics

The user should be able to calculate a

- financial ratio (total monthly loan payments divided by monthly net income) cash flow analysis (gross income minus expenditures). Keep a running total from month to month.

9. portfolio overview

Those who like to invest in the markets can choose this option to review their portfolio's performance.Include the categories

- stock/bond
- number of shares
- date purchased
- price
- commission

- total cost
- current value
- dividends
- net

Give totals where appropriate.

Notes:

You will need to decide on the appropriate format of these data.

Some suggestions are

- name(20 characters)
- social security number (9 characters)
- checking_balance (real)
- Savings_balance(real)
- list of transactions --code (2 digit integer), amount (8 digit real)

Your program must be menu driven.
Be sure that the user interface is consistent.
Once you have chosen a target language, you must get it approved by the instructor.

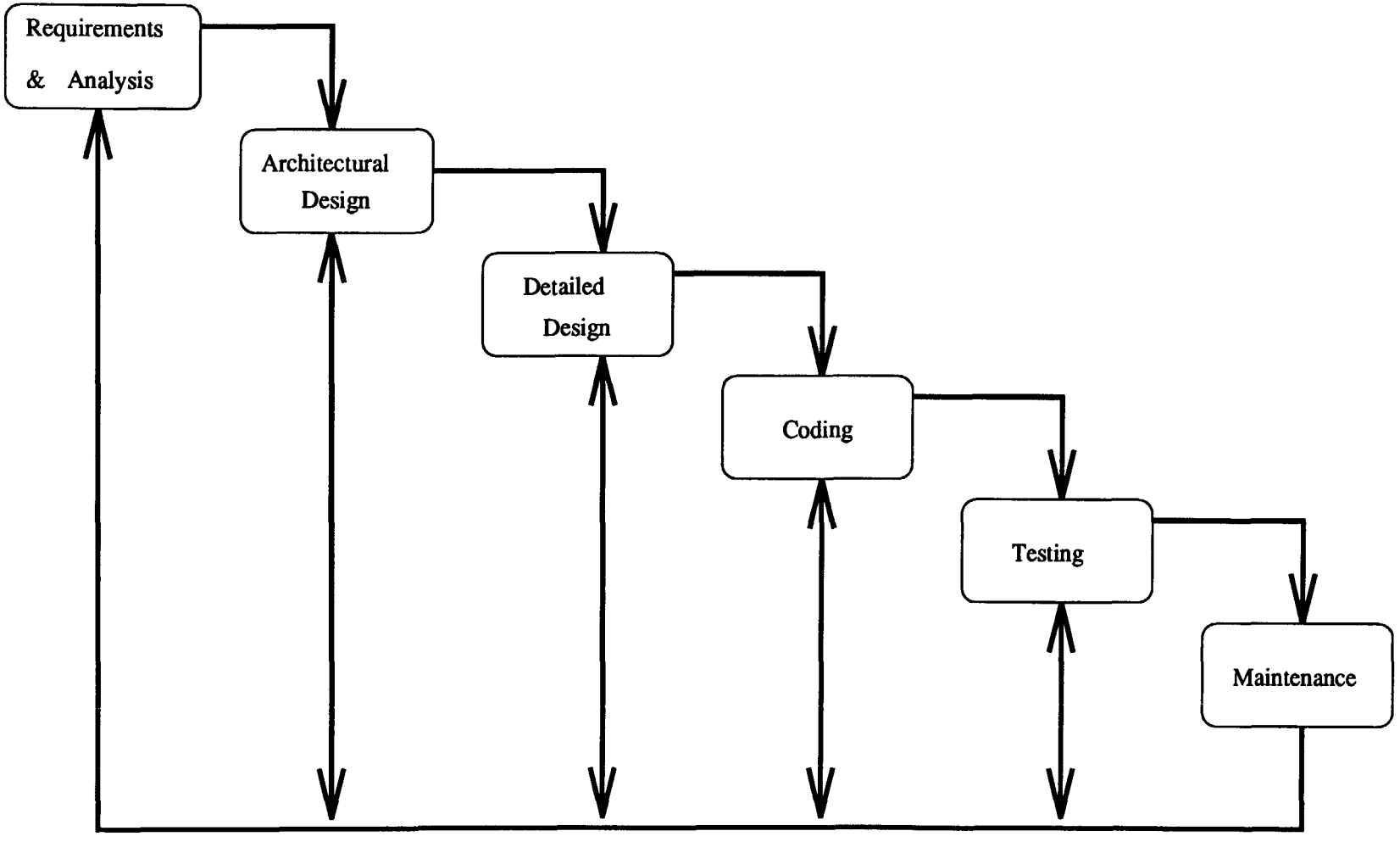
A user's manual will be your project's primary documentation. Write it for the computer novice. The program is to run on an IBM PC or compatible.

Use internal information as much as possible. This makes your system more efficient from the user's standpoint, as well as decreases the likelihood of input error.

WATERFALL MODEL

The students of this CS 680 class were to complete the project by using the waterfall model as a guideline for the software's life cycle. The software life cycle is the series of phases that a software product must undergo during its life. The waterfall model was first put forward by Royce (Royce, 1970). Figure 1 shows a diagram of the waterfall model (Pressman 1987). Discussion of the waterfall model follows.

Figure 1: Waterfall Model (the classic life cycle paradigm for software engineering)



REQUIREMENTS & ANALYSIS

The first phase in the software life cycle is determining the requirements for the system. The students of CS 680 asked as many questions as they thought necessary to help them determine the requirements of the system they were developing. This enabled them to have a clear understanding of the problem that they were to solve. A problem must be understood before it can be solved.

With only a small understanding of the desired software system, estimations of costs begin. Cost is divided into two parts, the dollar amount spent by the client and the time it will take the software engineers to develop the desired system.

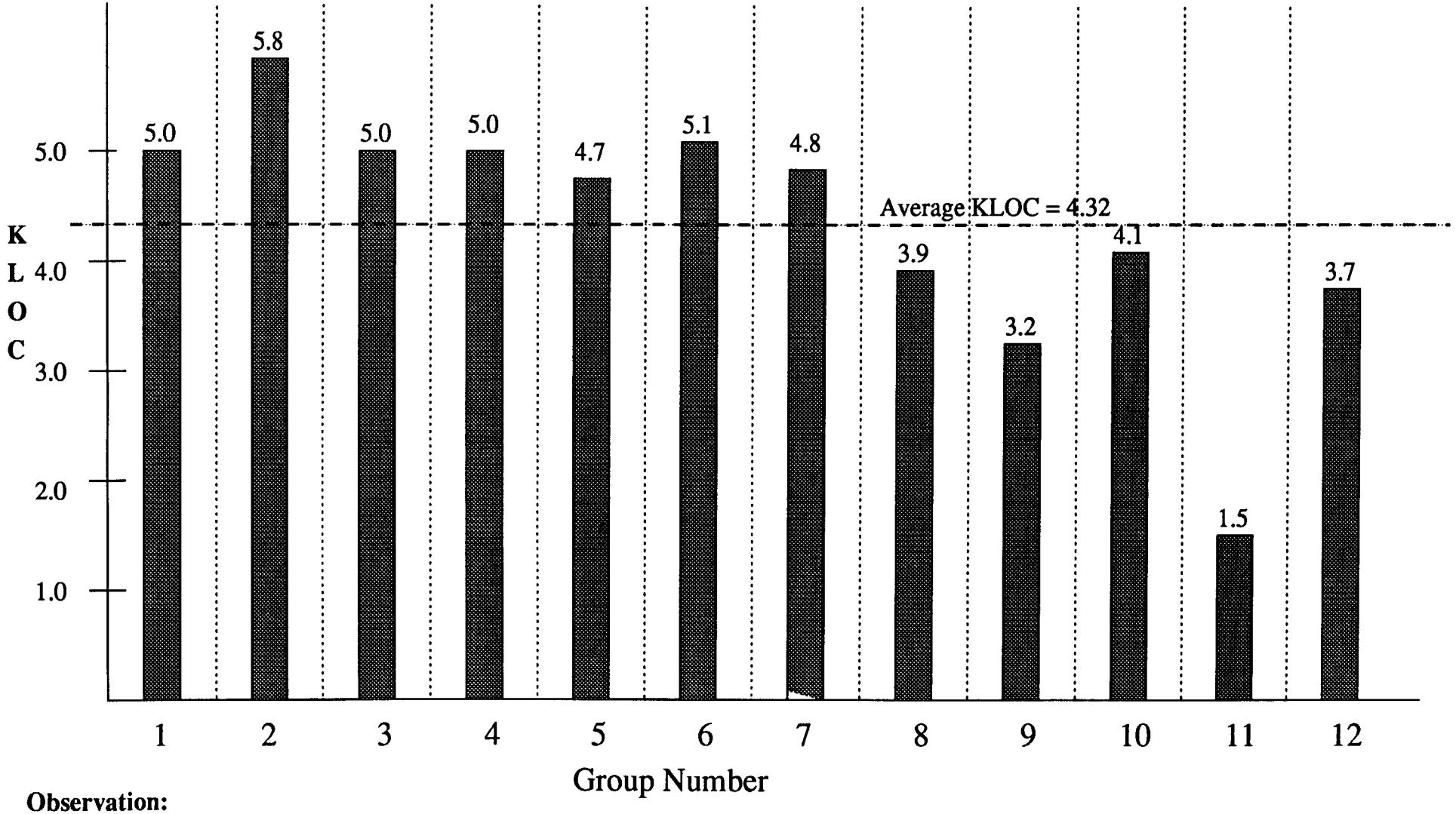
The budget is one important part in any software product. The client should know the anticipated cost of the product before development commences. If the development team underestimates the actual cost, it will lose money. Conversely, if the team overestimates, the client may decide that there is no point in having the product developed or give the job to another team whose estimate is more reasonable. Therefore, it is very important that cost be estimated accurately.

Time planning is another important part of software development. A deadline for the finished product should be established between the two parties. Dates for any other deliverables should also be agreed upon. If the team is unable to follow its schedule, it will lose its credibility and penalty clauses may be invoked. On the other hand, if the team overestimates the development time, the client might look for other options.

It is easy to say that an accurate estimation of time and money is necessary. However, estimating budget and time requirements with 100% accuracy is difficult because there are many variables to consider. The major difficulty is the estimation of human performance. All people work at different speeds and skill levels.

For the CS 680 course, students were to estimate the size of the product using the Delphi Technique which takes the average of expert opinions. The experts, in this case, were the team members. Following this estimation, the students ran COPSTAR, an effort estimation tool, to obtain estimates at the beginning of the project. Interestingly, estimations of size for the software system varied by the different teams. This can be seen in Figure 2.

Figure 2: Project size esitmations by different groups during the initial COPSTAR run



Observation:

* 6 groups estimated the product size to be 5.0 KLOC or more. Only 1 group estimated the product size below 3.0 KLOC.
* The difference between average LOC assumed and actual average LOC is 533. The actual average LOC is 3787.

COPSTAR was developed at the Purdue University Software Research Center by Dr. Samuel Conte and William Hsu. It is a tool to help management get early estimations on the number of personnel and effort needed to produce a piece of large software. Both the Cooperative Programming Model (COPMO) and the Constructive Cost Model (COCOMO) are used in this estimation.

COCOMO is actually a series of three models, ranging from a macroestimation model, which treats the product as a whole, down to a microestimation model, which deals with the product in detail. The major problem with COCOMO is that its most important input is the number of lines of code in the target product. If this estimate is incorrect, then every prediction of the model is affected. COPMO takes "into account not only the complexity of the software but also the complexity of interaction among the project team members" (Pfleeger 1991).

SPECIFICATIONS

Once the requirements are understood and estimations are made, the specifications for the software product should be documented. The specifications include general user requirements as well as hardware and software constraints. Data flow diagrams of the proposed system are also created during this phase.

A data flow diagram is designed to be close enough to user views so that a non-computer oriented user can understand what is happening. Those diagrams show the processes that occur and the data flows between them. (Lehman, 1991)

A process in a data flow diagram is represented by a circle or an oval. The process represents something that transforms a data flow. A data flow between processes is represented as an arrow. These arrows do not indicate control, as they would on a flow chart, but rather the flow of data. The contents of the data flows are documented in a data dictionary.

Terminators represent where data comes in from outside of the system and where data goes out from the system. A terminator can either be a source or a sink. A source is a provider of data, and a sink is a collector of data. Each is represented by a rectangle.

Sometimes data flows are not used directly by other bubbles but are stored for later use in another part of the system. The place where they are stored is known as a data store. In general, data stores are considered to be files.

One major benefit of data flow diagrams is that they graphically display what happens to a data item as it flows through the system. They can also help the analyst understand the existing system and identify problems associated with it. The most common problem is a dead-end data flow, where data goes in but no data is ever sent out (Lehman, 1991).

The data flow diagrams must be reviewed with the client. Until the client is fully satisfied, the data flow diagrams are refined and presented again to the client. Once the client is fully satisfied with the specifications, the design of the product begins. In the specifications and requirements analysis phase, WHAT the product has to do is described, whereas in the design phase, HOW the product has to do it is determined (Schach 1990).

ARCHITECTURAL DESIGN

The design phase, which is developed to make subsequent phases go more smoothly, is divided into two parts, architectural and detailed. In architectural design, architectural is used in the sense of taking the parts and assembling them into a meaningful whole. Therefore, architectural design describes the model for the software, whereas detailed design provides the details needed, complementing architectural design.

Although the requirements were the same for all teams in the CS 680 class, all twelve groups had a different design. This means that there are no right or wrong designs, only ones that lead to better software. The first step in architectural design is refining the data flow diagrams. Following these refinements, structure charts are developed by identifying transactions and transformations in the data flow diagrams.

A transformation in a data flow diagram is the set of processes involved in changing data. A transformation has input processes, output processes, and transformation processes. A transaction represents a center where different paths through the software system can be chosen. For example, a transaction could be a menu of possible options. After the processes in the data flow diagrams are identified as either a part of a transformation or a transaction, they are mapped to modules in a structure chart accordingly. Structure charts show the modular connections in a system and data that is passed between modules.

DETAILED DESIGN

Once the architectural design is completed, detailed design, the second phase of design starts. In this phase, structure charts are refined by applying design heuristics, such as low coupling and high cohesion to the structure charts. More details to each module are added at this time, such as describing module interfaces and writing pseudocode. Following the development and analysis of the structure chart and the approval of the client, the design phase ends and coding begins.

CODING

When a team starts coding, flaws in the design and faults in the specifications may appear. These faults are corrected, and the necessary changes are made in the specifications and design documents before further progress takes place in the development of the software product. For example, groups one and seven first decided to code using the C language. However, when they actually started coding, they found out that coding in a version of dBase would be easier for this type of application. Switching from C to dBase required changes in their specifications, data flow diagrams, structure chart, and data dictionary.

As Figure 1 shows, the waterfall model allows for the necessary revisions of the specifications and the design, at every phase of the software life cycle. All corrections should be properly documented so that no questions arise later about why a correction was made and who made it.

OVERVIEW OF ALL 12 PROJECTS

Students were free to choose any programming language. Eight groups chose dBase, one group chose Object-Vision, one group chose C and one group chose C and ParadoxENGINE. From Figure 3, we see that the average lines of code for the project is 3,787. Figure 4 displays the average lines of code per module for each team's project. Figure 5 shows how the project progressed from data flow diagrams to the structure chart to the actual number of coded modules. Because group eight did not finish, they are left out of some of the calculations. As you can see, all eleven completed projects look different from each other, once again stressing that there is more than one solution to a problem.

Figure 3: Total LOC of CS 680 Projects

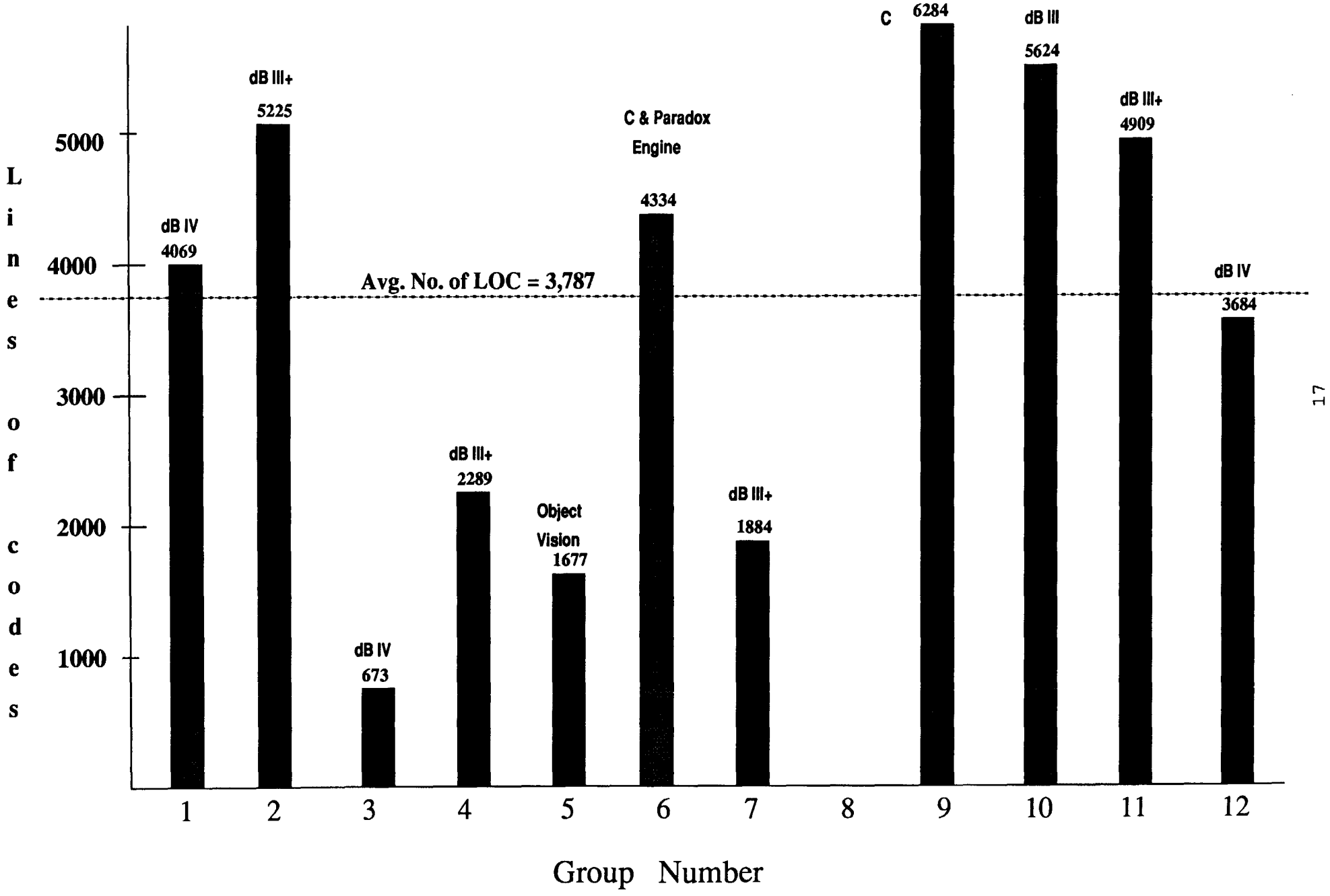
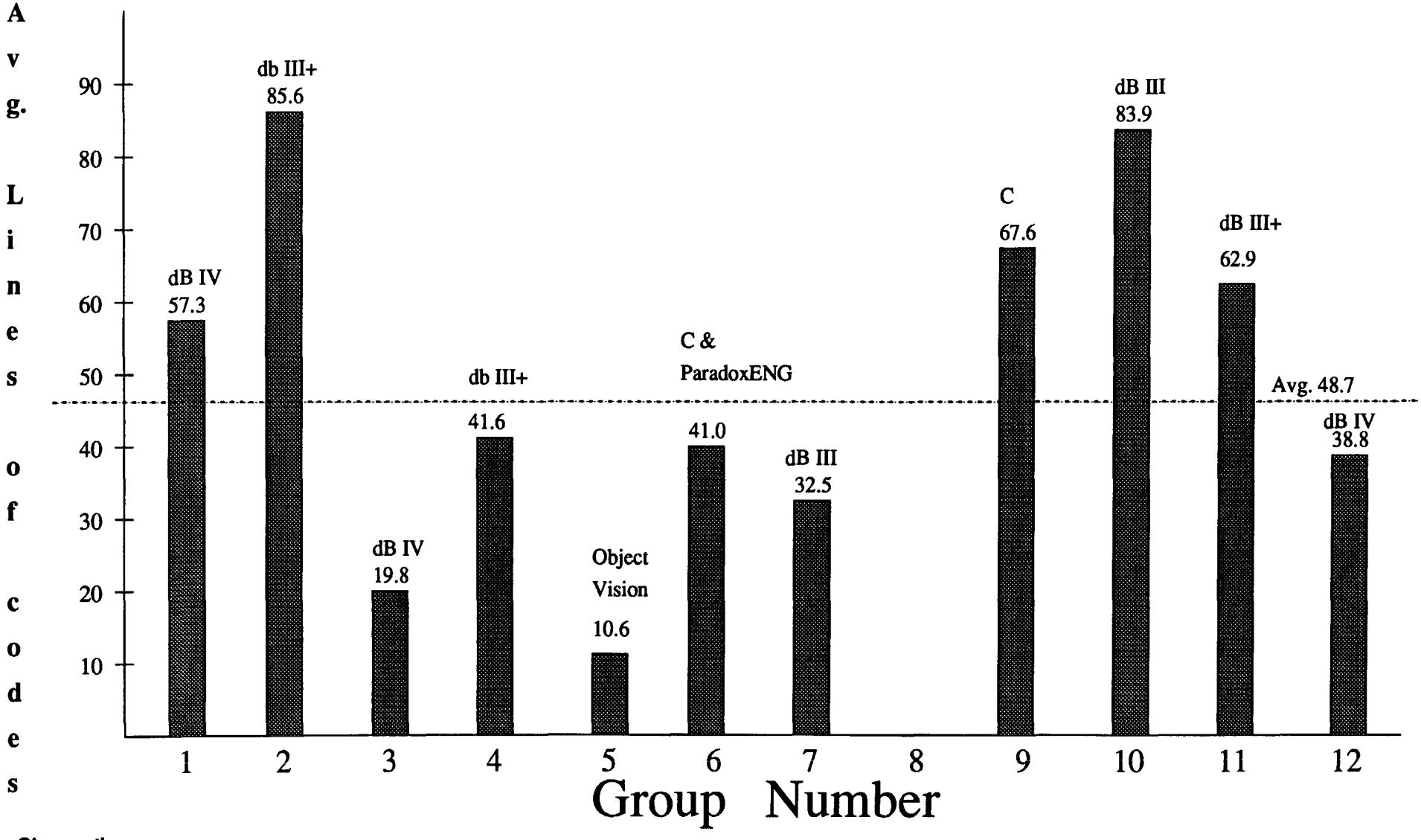


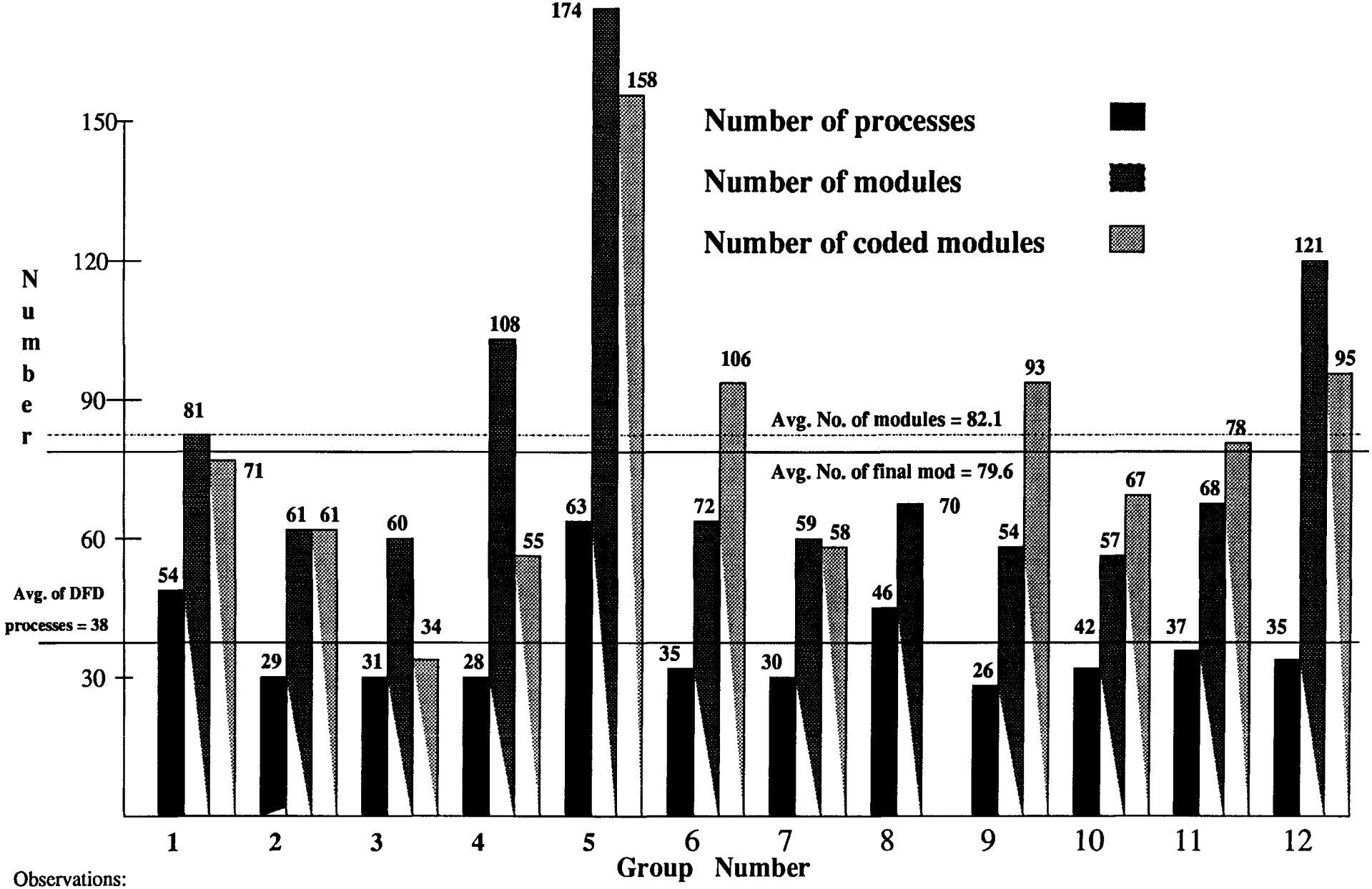
Figure 4: Average LOC per module for CS 680 Projects



Observations:

* **Gr. 5 used the forth generation language Object Vision resulting the least average LOC per module (10.6).**
* **For those systems using dB, the average LOC is 52.8 per module.**
* **Two groups used C language and the average LOC is 54.3 per module.**

Figure 5: Number of Processes/Modules in CS 680 Projects



Observations:

* From Data Flow Diagram (t1) to Structure Chart (t2), the number of modules have at least doubled in most cases.
* From Structure Chart (t2) to Final Structure Chart (t3), the number of modules have increased in most cases.

TESTING

Testing is not a process which only applies when the coding is done, but it is used throughout the software life cycle. For example, at the beginning, requirements and specifications must be checked for accuracy and completeness. At the design phase, careful validation at every stage must be done. Then, each module must be tested. Following integration, the product as a whole must be tested. Finally, after the product is tested for acceptance, it goes into operation mode and maintenance begins.

It is not a practice that a programmer should test a module he coded. It is a good idea to have a separate testing team whose purpose is to find flaws in the software. However, this does not mean that the original programmer should not debug his program or correct the errors. Before the testing begins, both the test data and the expected result should be recorded. After the test has been done, the actual results obtained should be recorded and compared with the expected results.

When does testing stop? Testing lasts throughout a software's life and is continuously being tested. "After a product has been successfully maintained for many years it may eventually lose its usefulness and be superseded by a totally different product. Or a product may still be useful but the cost of porting it to new hardware or of running it under a new operating system may be larger than the cost of constructing a new product, using the old one as a prototype. Finally, therefore, the software product is decommissioned and removed from service. Only at that point, testing can be stopped" (Schach 1990).

In CS 680 course, the due date was the indicator of when to stop integration testing. The software product was then given to the client Dr. Wayne Zage, the professor of the CS 680 course, so that he could perform acceptance testing. However, when the Design Metrics Research Team analyzes these projects, they are oftentimes tested some more. In the professional world, when the client agrees that the product meets the requirements and he accepts it, it is put in operation mode and any change becomes a part of maintenance.

MAINTENANCE

To make maintenance easier, changes to any aspect of the software system are documented and the following documents are necessary:

1. Specifications documents
2. Design documents
3. Code documents
4. Other documents
   such as - User Manual
   - Database Manual
   - Operations Manual

Sixty to seventy percent of software budgets are attributable to maintenance. Maintenance includes enhancements, which come about when the client changes the requirements. Once the requirements are changed, these changes naturally affect all subsequent phases. The waterfall model therefore can be said to be a dynamic model and the feedback loop plays an important role in this dynamism.

INTRODUCTION TO GROUP 6

For a detailed study of how design metrics can help in predicting errors, the Design Metrics Research Team chose the project of group six, which used the C language with a relational database language ParadoxENGINE, to analyze and draw conclusions.

First, the research team tested the software and found errors that had not been found and fixed by group six. Then those errors were traced to specific modules in the code. Following this, modules and processes were mapped from one time slice to the next. Then, design metrics D_c & D_i, were computed for three stages in the life cycle. Finally, algorithms were applied to the metric data to determine stress points. Then those identified stress points were compared with the actual locations of the errors found.

TESTING RESULTS

As mentioned previously, testing is a step in the software life cycle; therefore, the twelve projects were tested by their group members. However, as also stated, there is no definite end to testing a project. As a result, the research team again tested the project of group six.

There were 29 error modules detected containing 50 errors. These errors were classified as design, logical, minor, or critical errors. For our study, however, an error is counted as an error no matter how critical it was. For example, an error that crashed the system was counted the same as an error that allows you to input a nonexistent month.

After compiling a list of errors, tracked through the software by the menu selections, the research team traced the chosen menu options through the code until the module that contained the detected error was found. The line where the error occurred was actually located. If group six had still been doing their testing, they would have corrected these errors at this time.

MAPPING

After knowing where errors are located in the final code, the next step is to trace all the modules back to the first structure chart (t2) and then back to the data flow diagrams (t1). To get a better view of what was happening throughout this software life cycle, the research team developed a final structure chart (t3) from the final code. This was done because the first structure chart, developed by group six, was not reflective of the final code. Since the structure chart is developed before the code, updates in the code do not always get updated in the structure charts and data flow diagrams.

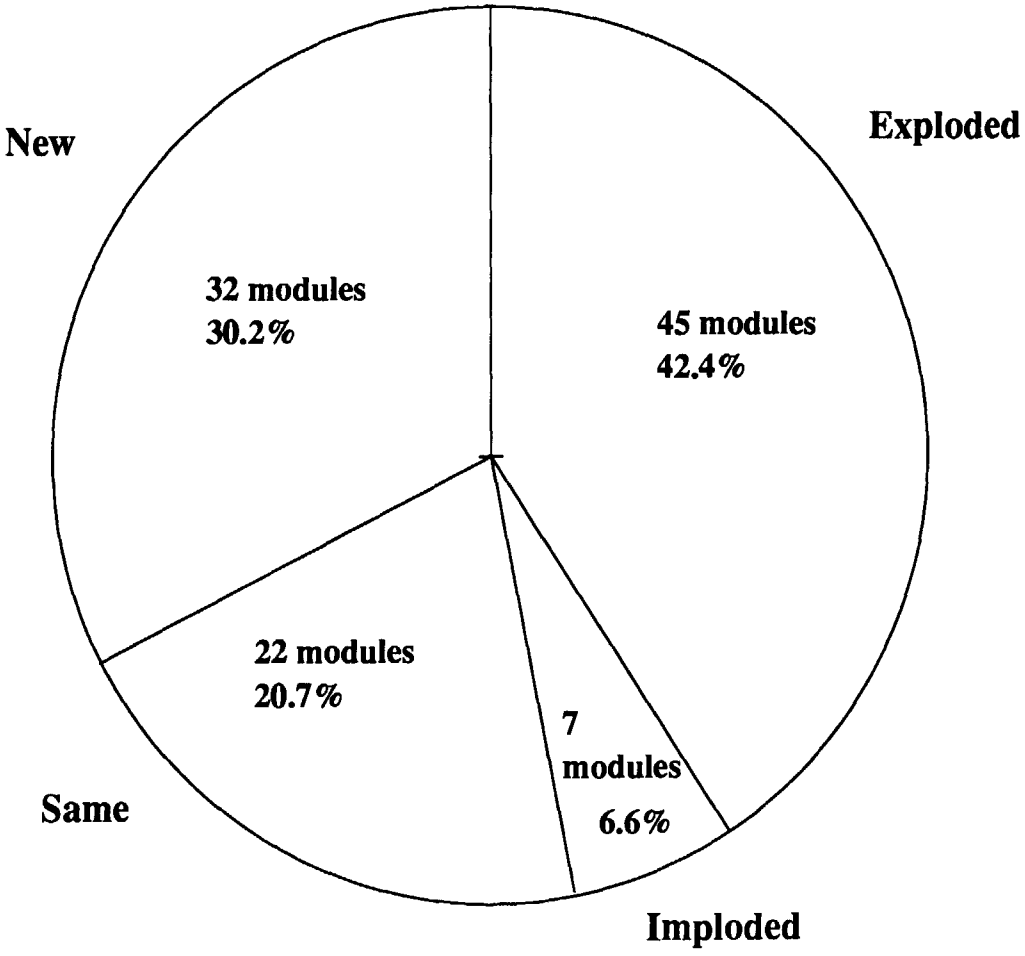
Because processes in the data flow diagrams did not always map with a one-to-one correspondence to the modules in the first structure chart, mapping was a difficult task. Likewise, the modules in t2 did not map one to one to modules in t3.

For the research team's study, modules from t1 (data flow diagrams) and t2 (first structure chart) to t3 (final structure chart) were evaluated. Several things occurred from one time slice to another. For example, new modules were added, some were deleted, some were imploded, some were exploded, and some stayed the same. As the design matured, more explosions than implosions occurred. Figure 6 shows the breakdown of categories of modules.

From the 106 modules in t3, 32 were new to this time slice. Forty-five modules had been created by exploding modules in t1 and t2. Seven modules had been formed by imploding modules from t1 and t2, and 22 modules stayed the same throughout all time slices.

The new modules accounted for 8 of the 29 error modules, while being responsible for 50% of the errors (Figure 7). This means that 25% of the new modules contained errors, and the average errors per module is 0.8. Although these modules contained one half of the total errors, it is not known in which error category these errors are classified. Because they are new, their errors might be minor errors that were just overlooked.

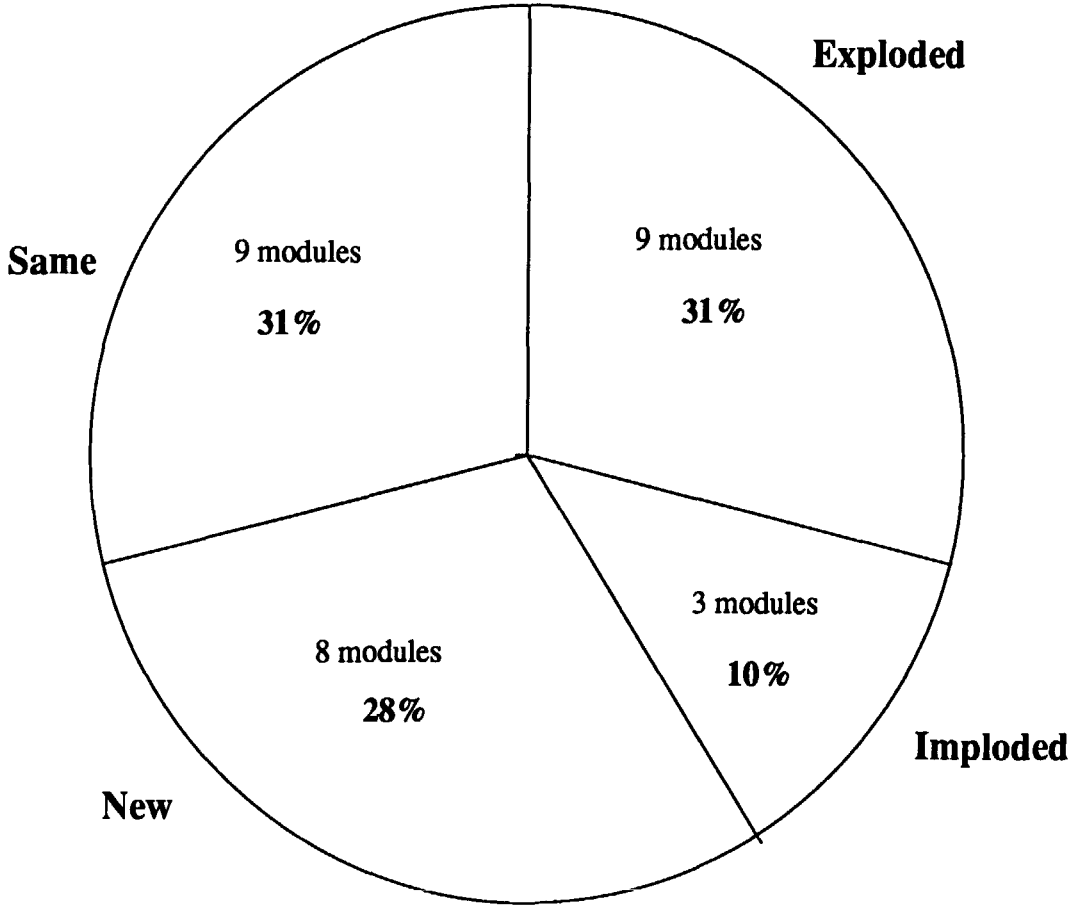
Figure 6: Total Number of Modules for Same, Exploded, Imploded and New Modules from t2 to t3 for Group 6 Project



Observations:

As the design matures, more modules were exploded.
13 modules from t2 were exploded into 45 modules in t3 resulting
42% of the total modules.

Figure 7: Number of Error Modules Occurring in Same, Exploded, Imploded and New Modules



Observations:

* There were 29 error modules in total. 8 modules in New Modules contained 25 errors.
* When the modules are exploded from one time slice to another, they are less likely to contain errors.
* New modules are likely to contain errors.

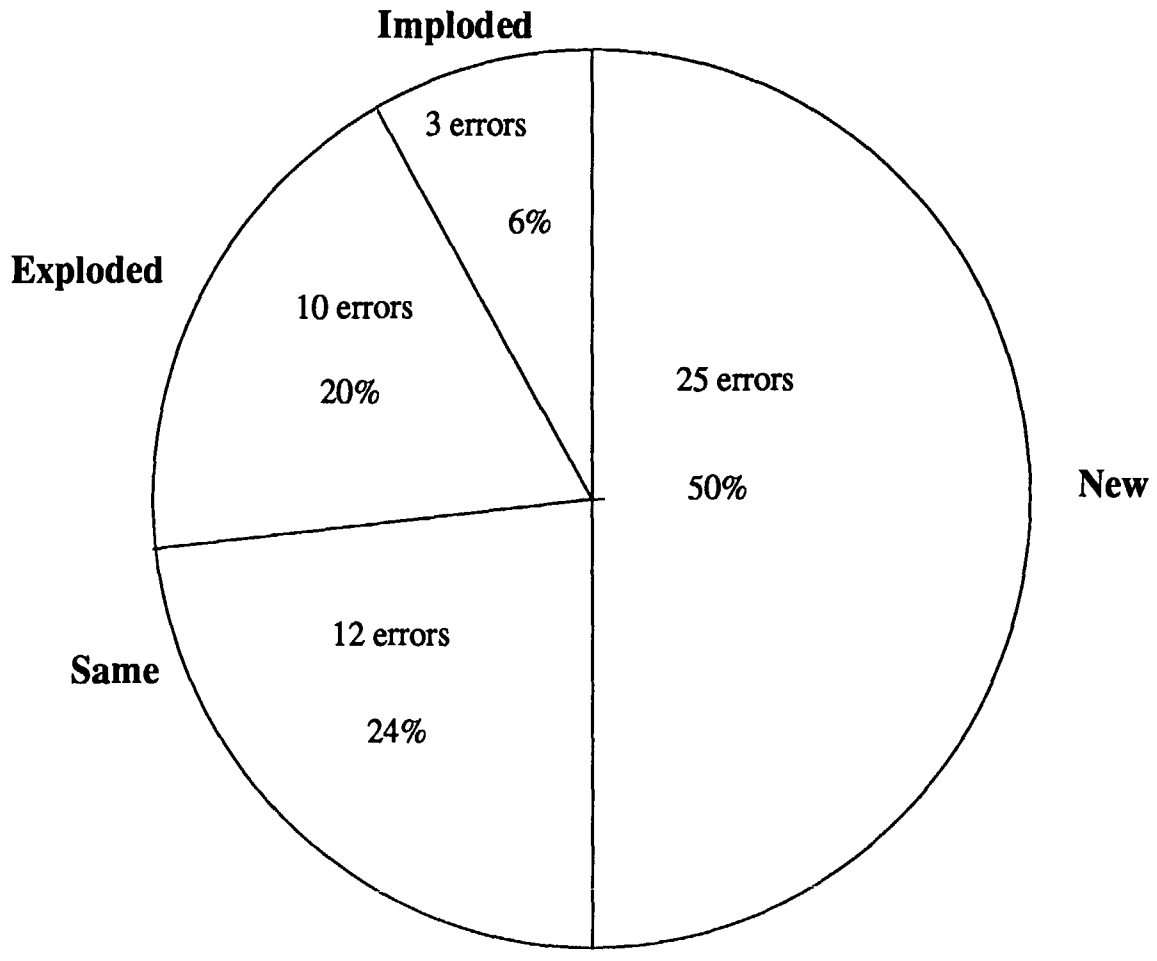
Twenty percent of the errors are attributed to the exploded modules. These errors were contained in 9 out of the 29 total error modules. Exploded modules are less likely to contain errors, because the functionality is being split into several modules. In the research team's study, each exploded module contained an average of 0.2 errors per module. Compared to the 0.8 errors per module for new modules, this is a small error rate.

Only 7% of the total modules are modules formed by implosion, and they contained only 6% of the total errors. However, 43% of these modules contained errors. Therefore, modules formed from implosion are likely to contain more errors, due to the combination of functionality.

Of the 106 modules, 22 of them stayed the same. Of these 22 modules, 9 contained errors, accounting for 24% of the total errors. Therefore, 41 percent of the modules that stayed the same contained errors. The average error rate for modules staying the same is 0.5 errors per module. Figure 7 shows the percentage of error modules accounted for by each type of module category. Figure 8 shows the number of errors in each category.

Now that we have all the processes and modules traced from t1 to t2 to t3, design metrics are calculated on each process and module at each time in the software life cycle.

Figure 8: Percentage of Errors Occurring in Different Types of Modules



Observations:

* Half of the errors occurred in new modules.
* When the modules are exploded from one time slice to another, they are less likely to contain errors.

D_e & D_i

In the research team's study D_e and D_i were calculated for processes in the data flow diagrams, for modules in the first structure chart and final structure chart. In order to make accurate counts, the research team had to learn how to do the counting.

D_e, representing the external complexity of a module, shows the modules interactions with other modules. It is composed of four components: datain, dataout, fanin and fanout.

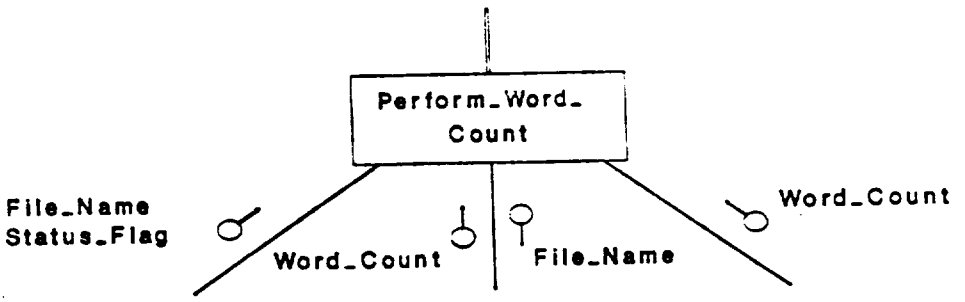
Datain consists of all variables coming into a module. These variables can be from the parent modules, the child modules, or global variables.

Dataout consists of all variables that get passed out of the module. If a variable comes in and is changed, it counts a dataout, including global variables if they are changed.

Fanin is calculated by counting the number of modules that call this module. Similarly, fanout is counted by counting the number of unique modules that a module calls. Standard library calls do not count.

These four components are applied in the following formula (Zage 1990):

$$D_e = (\text{datain} * \text{dataout}) + (\text{fanin} * \text{fanout})$$



$$D_e = (4 * 1) + (1 * 3) = 7$$

D_i, representing the internal complexity of a module shows the module's internal actions. It is composed of three components: central calls (CC), data structure manipulations (DSM), and input/output (I/O). Central calls consist of the total number of calls made to other modules.

Data structure manipulations are references to complex data types. Each reference counts as one, except in the case of pointers to other dsms. For example, x[i]->y[z]->a[b] counts as three data structure manipulations. Trees, tables, stacks, pointers, arrays, structures(records, fields), linked lists and queues are all examples of data structures.

Input/output counts consist of any input or output, excluding standard library calls. Examples include read, get, put, write, clrscreen and window statements. Statements such as setcolor do not count because they just set variables and the input/output is not really done until a clearscreen is executed.

The components are used to form the following equation where w1, w2, and w3 are weighting factors (Zage 1990):

$$D_i = w1(CC) + w2(DSM) + w3(I/O)$$

Example:

```
Procedure perform_word_count
 get_input(validated_filename, status flag)
 if (status_flag) is false then
  ERROR_TABLE[1]=FILENAME
  print("Error 1:file does not exists")
 else
  count_number_of_words(validated_filename,word_count)
   if (word_count)=-1 then
     ERROR_TABLE[2]="DOES NOT EXIST"
     print("Error 2:File is not a text file.")
    else
     process_output(word_count)
```

Italics = *CC*, LARGE = DSM, **Bold** = I/O

$D_i = (3 + 2 + 2) = 7$

The results of D_e for the specifications and architectural design phases are shown comparatively in Figure 9 and Figure 10. Figure 9 shows the total D_e for the first two phases studied, data flow diagrams and the initial structure chart. The total De for most groups increased with time. There are two reasons for this behavior.

Firstly, while calculating D_e for the data flow diagrams, D_e is only calculated for the primitive level processes. The overview, context, and any first-level or second-level processes were ignored. However, in the development of the structure chart, most context level processes transformed into dispatchers or control modules having higher D_e values. Therefore, even if the structure chart was created from the data flow diagrams without any refinements, the total D_e value will be higher.

Secondly, as the design matures, more details are added. This means new modules are introduced or processes from data flow diagrams are exploded into new modules. Adding new modules increases the total D_e value.

Figure 10 shows the average D_e values per process or module for each group. For nine out of the twelve groups, the average D_e decreased with time. This shows the data flow diagrams had relatively busy processes, whereas in the structure chart the modules were not communicating as much with each other. Therefore, the new modules added to the structure chart from the data flow diagrams probably had low D_e values causing the high D_e values to be spread out among the low De valued modules.

Figure 9: Total De of CS 680 Projects

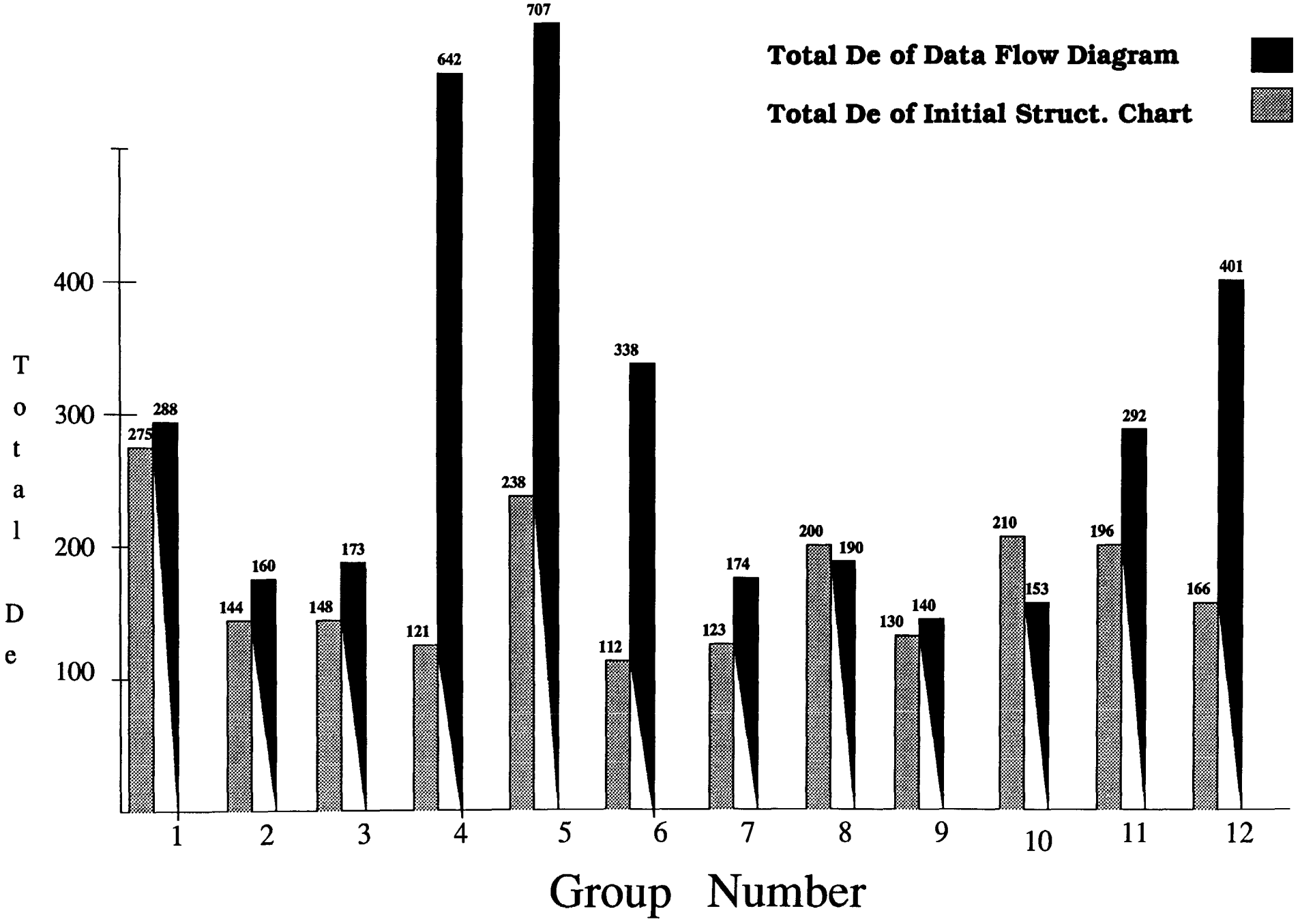
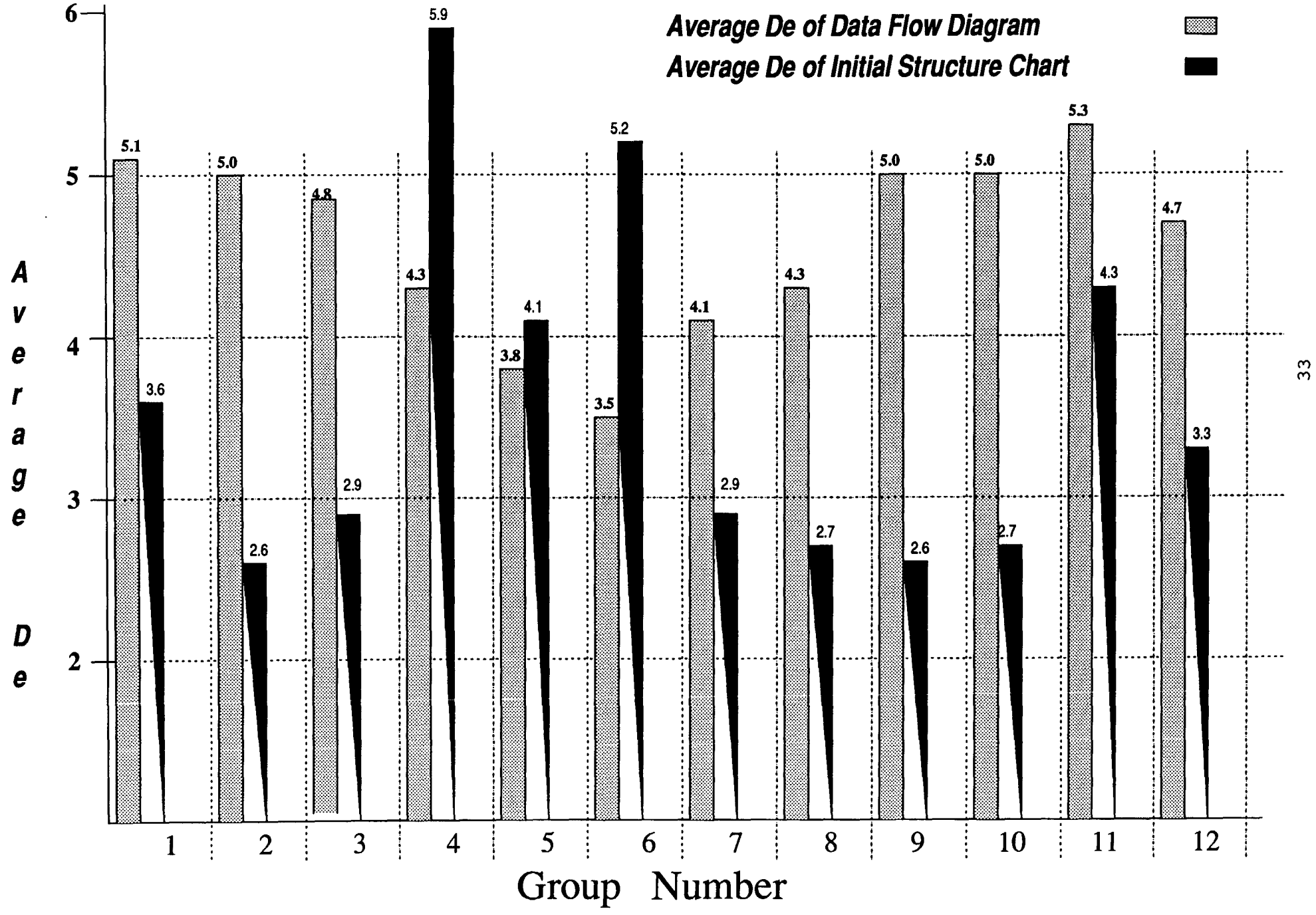


Figure 10: Average De of CS 680 Projects



CLASSIFYING MODULES

To see more of what was happening to the design of the group six project, the research team classified each module by its history. The categories exploded, imploded, new, and same were used. The research team already showed how the errors related to these categories, but now how D_e and D_i related to these categorized modules is shown.

To compare the metric values, the averages for D_e and D_i were computed for each category. Averages are used, because the modules are not evenly distributed among the categories. These values can be found in Table 1.

The new modules had the highest D_e average, yet the lowest D_i average. For this project, most of the modules introduced in t3 were utility or library modules. The modules naturally are called several times and have a low amount of activity inside. The low amount of activity inside can also be seen by the low average lines of code count. The fact that these modules are mostly utility modules explains the high D_e average, the external complexity, and the low D_i average, the internal complexity.

Typically, when a module is exploded, the new modules formed are spread out on different levels. Therefore, the D_e value for some modules is increased, but probably not significantly. In addition, the higher D_e value is spread out among these exploded modules when the average D_e is computed. Conclusively, exploded modules have the lowest D_e average.

Since the functionality of an exploded module is spread out among other modules, its D_i value would decrease. This theory holds for group six, since the exploded modules have the second lowest average D_i and lines of code counts.

Imploded modules have the highest metric values for tow metrics, lines of code and D_i. Looking at it simply, when two or more modules are transformed into one, you would basically add all the metrics together. Thus, the internal metrics would increase. However, the external metric D_e would not increase by as much, because there will usually be some overlap in the contributing modules' call structures. For example, if the imploded modules call the same module, this call would only be counted once.

Table 1: Metric Averages for New, Same, Imploded & Exploded Modules in Group 6 Project

Module Type	Number of Modules	Avg. LOC per module	Average Di	Average De
New	32	33	16	157.8
Same	22	47	34	51.0
Imploded	7	64	43	57.4
Exploded	45	40	32	21.6
TOTAL	106			

EVALUATIONS & ALGORITHMS

In the research team's study, the focus was on D_e counts and how they changed over time. The research team calculated D_e values in order to identify modules as stress points. These modules, according to previous studies performed on design metrics, are more likely to contain errors. Twenty-five algorithms were created and applied to the data in order to identify the greatest number of errors and error modules. These algorithms are displayed in Table 2.

To begin, an algorithm, algorithm 1, was used. Having been used in the past, this algorithm has proven to be successful. In this algorithm, the average D_e value of the modules in t3 is calculated with the standard deviation identified. Those modules with a D_e metric value one standard deviation above the mean are considered stress points. Using the x-less algorithm which removes the highest D_e value from the calculation of the mean and standard deviation, more modules are identified as stress points. This algorithm will be referred to as the standard algorithm and displayed as ti-x-less, where i represents the time slice number and x represents the number of times the x-less algorithm was applied.

In order to determine how good of a predictor of errors an algorithm was, the percentages of errors and error modules detected were calculated, along with the percentage of total modules that had to be considered to achieve the results.

Algorithms 2 through 6 are all variations of applying the standard algorithm to the different time slices. For example, algorithm 2 used only the standard algorithm on all three time slices. Algorithm 7 applies the standard algorithm to time slices 1 and 2 and includes all modules new to t3 as stress points. Although this algorithm identified over 50% of the total errors, too many modules have to be considered to achieve these results. Programmers and designers want to look at as few modules as necessary in order to find the most errors. To compensate for this, algorithms 8 through 11 are variations of applying the standard algorithm to the set of modules new to t3.

As mentioned before, as a process or module develops it might be deleted, exploded, imploded, added, or kept the same. For modules identified as stress points in earlier time slices, in this case t1 and t2, their consideration in t3 varied based on their history.

Table 2: Identifying the Best Predictor of Error-Prone Modules

S. No	Algorithm used	Mods checked		error mods		# of errors		% of errors/ % of total mods	% of error mods/% of total mods	Alg. Rank
		nos.	%	nos.	%	nos.	%			
1	De on t3	9	8.5	6	21	19	38	4.5	2.5	1
2	t1-0-less; t2-0-less; t3-0-less	12	11.0	6	21	19	38	3.5	1.9	5
3	t1-0-less; t2-0-less; t3-1-less	13	12.3	6	21	19	38	3.1	1.6	10
4	t1-0-less; t2-0-less; t3-2-less	14	13.2	6	21	19	38	2.9	1.7	10
5	t1-0-less; t2-0-less; t3-3-less	16	15.1	8	27	21	42	2.8	1.8	10
6	t1-1-less; t2-0-less; t3-3-less	25	23.4	11	38	25	50	2.1	1.6	13
7	t1-0-less; t2-0-less; t3 (ALL NEW)	41	38.7	11	38	28	56	1.4	1.0	16
8	t1-0-less; t2-0-less; t3(NEW)-0-less	11	10.4	5	17	18	36	3.6	1.6	7
9	t1-0-less; t2-0-less; t3(NEW)-1-less	12	10.4	6	21	19	38	3.6	2.0	4
10	t1-0-less; t2-0-less; t3(NEW)-2-less	13	12.3	6	21	19	38	3.1	1.7	9
11	t1-0-less; t2-0-less; t3(NEW)-3-less	16	15.1	6	21	19	38	2.5	1.4	14
12	t1-0-less(exp); t2-0-less(exp); t3-0-less	10	9.4	6	21	19	38	4.0	2.2	2
13	t1-1-less; t2-0-less; t3-0-less	14	13.2	9	31	23	46	3.5	2.3	3
14	t1-0-less(exp); t2-0-less(exp); t3-1-less	11	10.4	6	21	19	38	3.6	2.0	4
15	t1-0-less(exp); t2-0-less(exp); t3-2-l	12	11.3	6	21	19	38	3.4	1.9	6
16	t1-0-less(exp); t2-0-less(exp); t3-3-less	14	13.2	8	28	21	42	3.2	2.1	5
17	t1-0-less(exp);t2-0-less(exp);t3(new)-0-less	9	8.5	5	17	18	36	4.2	2.0	3
18	t1-0-less(exp);t2-0-less(exp);t3(new)-1-less	10	9.4	6	21	19	38	4.0	2.2	2
19	t1-0-less (exp);t2-0-less(exp);t3(new)-2-less	11	10.4	6	21	19	38	3.6	2.0	4
20	t1-0-less(exp);t2-0-less(exp);t3(new)-3-less	14	13.2	6	21	19	38	2.9	1.6	11
21	t3 (categorized)-0-less	11	10.4	5	17	18	36	3.5	1.6	8
22	t1-0-less; t2-0-less	9	8.5	3	10	3	6	0.7	1.2	18
23	t1-1-less; t2-0-less	18	17.0	6	21	7	14	0.8	1.2	17
24	t1(exploded)-0-less; t2(exploded)-0-less	7	6.6	3	10	3	6	0.9	1.5	15
25	t1(exploded)-1-less; t2(exploded)-0-less	11	10.4	6	21	7	14	1.3	2.0	12

For deleted modules, the decision is easy. You cannot consider those modules in t3 that do not exist. Modules that are imploded are still carried forward as stress points, because if one of the contributing modules was identified as a stress point, chances are the resulting imploded module is even more complex. However, implosion could reduce the number of modules to be considered in a later time slice. For example, if three modules were identified as stress points in t2, but in t3 these modules were imploded into one, only that one module in t3 has to be considered.

Modules staying the same that were identified as stress points in an earlier time slice can either be carried on as stress points or can be dropped from consideration. If they are dropped, we assume that if they are still stress points in the later time slice they will be picked up by the standard algorithm.

When a module is identified as a stress point that later is exploded, you can consider all the modules it became or only consider a subset of these modules. In algorithms 2 through 11, all the resulting modules were considered as stress points. However, probably not all the resulting modules will contain errors, if any of them at all. Therefore, only a subset of these exploded modules was chosen to consider. Since D_e has been a good predictor of error-prone modules in the past, the modules with the highest D_e value among the modules resulting from the same exploded module will be carried on as stress points. If two or more modules are tied for the highest D_e value, all of them will be considered. This algorithm of only carrying forward modules with the highest D_e value that were formed through an explosion will be referred to as exploded algorithm and represented as (exp) in Table 2.

Algorithms 12 through 16 show variations of the above mentioned conditions for exploded, imploded, same, and deleted modules, where modules staying the same are carried forward into consideration. Algorithms 17 through 20 apply the same conditions, except variations of the new modules in t3 are used.

Algorithm 21 is computed by categorizing the modules in t3 as discussed in the classifying modules section. The standard algorithm is then applied to each of these subsets. Table 3 displays more details of this algorithm.

Table 3: Explanation of Algorithm 21

Module Type	Number of Modules	% of errors	Avg. LOC per mod.	Average Di	Average De	Average De		Std. Dev. of De	Stress pt cut off	Number of modules			errors
						error mod	errorless			high.	hit	miss	
New	32	50	33	16.0	157.8	535	32	451.9	610	2	2	0	15
Same	22	24	47	34.4	51.0	25	69	93.7	145	2	0	2	0
Imploded	7	6	64	41.3	57.4	91	32	53.4	111	1	1	0	1
Exploded	45	20	40	32.0	21.6	37	18	26.8	49	6	2	4	2
TOTAL	106	100								11	5	6	18

The goals of the Design Metrics Research Team include identifying error-prone modules early in design. Algorithms 22 through 25 are variations of only considering the first two time slices which were created very early in the development process.

In order to determine what constitutes a good algorithm, the research team first divided the percentage of error modules detected by the percentage of total modules considered and then divided the percentage of errors detected by the percentage of total modules considered. The higher these numbers were, the better the algorithm. These two values were then added to determine the best algorithm based on both criteria. The algorithms were ranked accordingly. The best algorithm was the first one, where the standard algorithm was applied to t3. This is not surprising, since the metrics were computed from the final code.

The second best algorithms were 12 and 18. Algorithm 12 used the standard algorithm on all three time slices with 0-less and the exploded algorithm on modules identified in t1 and in t2 as stress points. This result was good, since the metrics team wanted to relate stress points throughout time slices to error modules. Algorithm 18 wascalculated similarly to algorithm 12 except in t3, the standard algorithm was applied to only the new modules with 1-less.

The algorithms that only use time slices t1 and t2 did not do too badly considering they could be calculated at such an early phase in the development process.

From these twenty-five algorithms the research team has designed a new algorithm that combines the aspects of the best algorithms. The algorithm applies the standard algorithm to the first time slice resulting in a set of stress points. This set will later be manipulated in the subsequent time slice and become the set of stress points identified in the previous time slice. When the standard algorithm is applied to all the modules in the current time slice, a set of stress points occurs. The standard algorithm is then applied to the modules new to this time slice creating another set of stress points. The set of stress points identified in the previous time slice is reduced by examining the history of the modules from the previous time slice to the current time slice. Only modules that stayed the same, imploded modules, and modules with the highest D_e value for exploded modules are included in the set of stress points identifies in the previous time slice. The union of these three sets results in the final set of stress points for the current time slice. When the next time slice is introduced, this final set becomes the set of

stress points identified in the previous time slice, and the determination of the other sets continues. Results for this algorithm are currently being studied. Because of the nature of the development of this algorithm, it will probably be very good.

FUTURE DIRECTIONS

The research team's next step is to compare these results to other graduate projects to see if the results hold. Because industrial software may differ from the graduate projects, studies this year and next will be done to compare these results with results obtained from analyzing industrial software. Design recommendations will be made to alter the software development process. If a module is identified as a stress point during design, the designers will decide what to do with that module based on the research team's predictions of its error-proneness. For example, the stress point module might be exploded or another level of modules might be added. The Design Metrics Research Team wants to provide design rules in order to improve the quality of the software product.

Also, the research team will be looking more closely at D_i and finding new ways to improve D_e calculations. Currently, analyzers and programs to calculate the metrics automatically so they no longer have to be calculated manually are being created. Doing so will yield more accurate counts and quicker results. These analyzers will be implemented using Lex and Yacc to retrieve the tokens and the C language to compute the metrics.

REFERENCES

1. Keuffel, Warren. "Software Engineering's Foundations". Computer Language. Graw-Hill Book Company. New York. 1987. pps 25-29.

2. Lehman, John A. Systems Design In the Fourth Generation. John Wiley & Sons. New York. 1991.

3. Pfleeger, Shari Lawrence. Software Engineering. The Production of Quality Software. Macmillan Publishing Co. 1991.

4. Pressman, Roger S. Software Engineering: A Practitioner's Approach. McGraw-Hill Publishing Co. New York. 1987.

5. Royce, W. W. "Managing the Development of Large Software Systems: Concepts and Techniques." Proceedings of WestCon. August 1970.

6. Schach, Stephen R. Software Engineering. Irwin. Boston. 1990.

7. Zage, Wayne M and Dolores M. Zage. "Design Metrics Poster Presentation." SERC Semi-Annual Meeting. Gainesville, FL. November 1990.